Assembleur

Un **langage d'assemblage** ou **langage assembleur** est, en programmation informatique, un langage de bas niveau qui représente le langage machine sous une forme lisible par un humain. Les combinaisons de bits du langage machine sont représentées par des symboles dits « mnémoniques » (du grec *mnêmonikos*, relatif à la mémoire), c'est-à-dire faciles à retenir. Le programme assembleur convertit ces mnémoniques en langage machine, ainsi que les valeurs (écrites en décimal) en binaire et les libellés d'emplacements en adresses, en vue de créer par exemple un fichier objet ou un fichier exécutable.

Dans la pratique courante, le même terme *assembleur* est utilisé à la fois pour désigner le langage d'assemblage et le programme assembleur qui le traduit. On parle ainsi de « programmation en assembleur ».

La traduction une fois pour toutes par beaucoup d'interpréteurs de chaque nom de variable rencontré dans une instruction (évoluée) par la position mémoire associée et de chaque constante (écrite par l'utilisateur en décimal) en binaire est typique d'une opération d'assemblage, bien que le nom d'assembleur ne soit pas couramment utilisé dans ce cas précis.

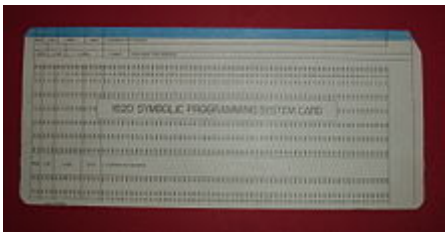
Sommaire

Histoire

Les programmes de l'EDSAC (1949)[1], premier calculateur à programmes enregistrés, étaient rédigés en utilisant des mnémoniques alphabétiques d'une lettre pour chaque instruction. La traduction était alors faite à la main par les programmeurs, une opération longue, fastidieuse et entachée d'erreurs.

Le premier programme assembleur a été écrit par Nathaniel Rochester pour l'IBM 701 (le premier ordinateur commercialisé par IBM) en 1954.

Les langages d'assemblages ont éliminé une grande partie des erreurs commises par les programmeurs de la première génération d'ordinateurs, en les dispensant de mémoriser les codes numériques des instructions et de faire des calculs d'adresses. La programmation en assembleur était alors utilisée pour écrire toutes sortes de programmes.


Carte formatée pour être perforée pour l'assembleur de l'IBM 1620.

Dans les années 1970-80, l'utilisation de l'assembleur pour écrire des applications a été très largement supplantée par l'emploi de langages de programmation de haut niveau : Fortran, COBOL, PL/I, etc. : la puissance des machines le permettait et consacrer quelques minutes de temps machine à une compilation pour économiser quelques heures de temps de programmeur était une opération rentable, même si les compilateurs de l'époque fournissaient un code moins performant (plus volumineux et souvent plus lent). Par ailleurs, ces langages de haut niveau permettaient de s'affranchir de la dépendance à une architecture matérielle unique.

Les systèmes d'exploitations ont été écrits en langage d'assemblage jusqu'à l'introduction de MCP de Burroughs, en 1961, qui était écrit en ESPOL, dialecte d'Algol.

L'assembleur est revenu quelque peu en faveur sur les premiers micro-ordinateurs, où les caractéristiques techniques (taille mémoire réduite, puissance de calcul faible, architecture spécifique de la mémoire, etc.) imposaient de fortes contraintes, auxquelles s'ajoute un facteur psychologique important, l'attitude « hobbyiste » des premiers utilisateurs de micro-ordinateurs, qui ne se satisfaisaient pas de la lenteur des programmes écrits avec le BASIC interprété généralement fourni avec l'ordinateur.

Des gros programmes ont été écrits entièrement en assembleur pour les micro-ordinateurs, comme le système d'exploitation DOS de l'IBM PC (environ 4000 lignes de code), et le tableur Lotus 1-2-3 (son rival Multiplan, qui existait déjà sous CP/M, était écrit en C). Dans les années 1990, c'était aussi le cas pour la plupart des jeux pour consoles vidéo (par exemple pour la Mega Drive ou la Super Nintendo).

Particularités de l'assembleur

Un langage spécifique à chaque processeur

Le langage machine est le seul langage qu'un processeur puisse exécuter. Or chaque famille de processeurs utilise un jeu d'instructions différent.

Par exemple, un processeur de la famille x86 reconnaît une instruction du type

```
10110000 01100001
```

En langage assembleur, cette instruction est représentée par un équivalent plus facile à comprendre pour le programmeur :

```
  movb $0x61,%al
```

(10110000 = movb %al
01100001 = $0x61)

Ce qui signifie : « écrire le nombre 97 (la valeur est donnée en hexadécimal : $61_{16} = 97_{10}$) dans le registre AL ».

Ainsi, le langage assembleur, représentation exacte du langage machine, est spécifique à chaque architecture de processeur. De plus, plusieurs groupes de mnémoniques ou de syntaxes de langage assembleur peuvent exister pour un seul ensemble d'instructions, créant ainsi des macro-instructions.

Désassemblage

La transformation du *code assembleur* en langage machine est accomplie par un programme nommé assembleur. L'opération inverse, à savoir retrouver l'assembleur équivalent à un morceau de code machine, porte un nom : il s'agit du *désassemblage*.

Contrairement à ce que l'on pourrait penser, il n'y a pas toujours de correspondance un à un (une bijection) entre le code assembleur et le langage machine. Sur certains processeurs, le désassemblage peut donc donner un code dont la compréhension est très difficile pour un humain tout en restant parfaitement compilable par un ordinateur. L'impossibilité d'un désassemblage peut avoir diverses raisons : usage de code auto-modifiant, instructions de taille variables, impossibilité de faire la différence entre code et données, etc. (code impénétrable)

Qui plus est, de nombreux éléments présents dans un code assembleur sont perdus lors de sa traduction en langage machine. Lors de la création du code en assembleur, le programmeur peut affecter des noms aux positions en mémoire, commenter son code, utiliser des macro-instructions ou utiliser du code généré sous conditions au moment de l'assemblage. Tous ces éléments sont réduits lors de l'assemblage à ce qui est strictement nécessaire pour la machine et n'apparaissent donc pas clairement lors du désassemblage : par exemple, une position en mémoire n'est repérée que par son adresse numérique ou par un offset.

Instructions machine

Certaines opérations fondamentales sont disponibles dans la plupart des jeux d'instructions.

- déplacement dans la mémoire :
  - chargement d'une valeur dans un registre ;
  - déplacement d'une valeur depuis un emplacement mémoire dans un registre, et inversement ;
- calcul :
  - addition, ou soustraction des valeurs de deux registres et chargement du résultat dans un registre ;
  - combinaison de valeurs de deux registres suivant une opération booléenne (ou opération bit à bit) ;
- modification du déroulement du programme :
  - saut à un autre emplacement dans le programme (normalement, les instructions sont exécutées séquentiellement, les unes après les autres) ;
  - saut à un autre emplacement, mais après avoir sauvegardé l'emplacement de l'instruction suivante afin de pouvoir y revenir (point de retour) ;
  - retour au dernier point de retour ;
- comparaison :
  - comparer les valeurs de deux registres.

Et on trouve des instructions spécifiques avec une ou quelques instructions pour des opérations qui auraient dû en prendre beaucoup. Exemples :

- déplacement de grands blocs de mémoire ;
- multiplication, division ;
- arithmétique lourde (sinus, cosinus, racine carrée, opérations sur des vecteurs) ;
- application d'une opération simple (par exemple, une addition) à un ensemble de données par l'intermédiaire des extensions MMX ou SSE des nouveaux processeurs.

Directives du langage assembleur

En plus de coder les instructions machine, les langages assembleur ont des directives supplémentaires pour assembler des blocs de données et affecter des adresses aux instructions en définissant des étiquettes ou labels.

Ils sont capables de définir des expressions symboliques qui sont évaluées à chaque assemblage, rendant le code encore plus facile à lire et à comprendre.

Ils ont habituellement unlangage macro intégré pour faciliter la génération de codes ou de blocs de données complexes.

Exemples simples

Voici quelques exemples simples :

- en syntaxe AT&T (écrits pour l'assembleur GNU (GAS) pour Linux) ;
- utilisant le jeu d'instructionsi386 ;
- à utiliser comme suit :

```
$ gcc foo.S -c -o foo.o
$ ld foo.o -o foo
$ ./foo
```

Afficher Bonjour

(les commentaires se trouvent après les points-virgules)

```
str:
 .ascii "Bonjour\n"
 .global _start

_start:
movl $4, %eax
movl $1, %ebx
movl $str, %ecx
movl $8, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80

;Compilation:
;as code.s -o code.o
;ld code.o -o code

;Execution:
;./code
```

Lire le clavier (16 caractères max) puis l'afficher

```
# define N 16

        .global _start

        .comm   BUFF    , N

_start: mov     $3      , %eax
        mov     $0      , %ebx
        mov     $BUFF   , %ecx
        mov     $N      , %edx
        int     $0x80

        mov     %eax    , %edx
        mov     $4      , %eax
        mov     $1      , %ebx
        mov     $BUFF   , %ecx
        int     $0x80

        mov     $1      , %eax
        mov     $0      , %ebx
        int     $0x80
```

Exemples simples, syntaxe Intel x86

Voici les mêmes exemples, avec quelques différences :

- en syntaxe Intel x86, écrit pour l'assembleur NASM ;
- utilisant le jeu d'instructions i386 ;
- à utiliser comme suit :

```
$ nasm -f elf foo.asm
$ ld -o foo foo.o -melf_i386
$ ./foo
```

Afficher Bonjour

(les commentaires se trouvent après les points-virgule)

```
section .data                                   ; Variables initialisées
        Buffer:           db 'Bonjour', 10      ; En ascii, 10 = '\n'. La virgule sert à concaténer les
chaines
        BufferSize:     equ $-Buffer            ; Taille de la chaine

section .text                                   ; Le code source est écrit dans cette section
        global _start                           ; Définition de l'entrée du programme

_start:                                         ; Entrée du programme

        mov eax, 4                              ; Appel de sys_write
        mov ebx, 1                              ; Sortie standard STDOUT
        mov ecx, Buffer                         ; Chaine à afficher
        mov edx, BufferSize                     ; Taille de la chaine
        int 80h                                 ; Interruption du kernel

        mov eax, 1                              ; Appel de sys_exit
        mov ebx, 0                              ; Code de retour
        int 80h                                 ; Interruption du kernel
```

Lire le clavier (64 caractères max) puis l'afficher

```
section .bss                                    ; Section des variables non-initialisees
        Buffer:   resb 64                       ; Reservation de 64 blocs (octets ?) memoire pour la
variable où sera stockee l'entree de l'utilisateur
        BufferSize:         equ $-Buffer        ; taille de cette variable

section .text                                   ; Section du code source
        global _start

 _start:                                        ; Entree du programme

        mov eax, 3                              ; Appel de sys_read
        mov ebx, 0                              ; Entree standard STDIN
        mov ecx, Buffer                         ; Stockage de l'entree de l'utilisateur
        mov edx, BufferSize                     ; Taille maximale
        int 80h                                 ; Interruption du kernel

        mov eax, 4                              ; Appel de sys_write
        mov ebx, 1                              ; Sortie standard STDOUT
        mov ecx, Buffer                         ; Chaine à afficher
        mov edx, BufferSize                     ; Taille de la chaine
        int 80h                                 ; Interruption du kernel

        mov eax, 1                              ; Appel de sys_exit
        mov ebx, 0                              ; Code de retour
        int 80h                                 ; Interruption du kernel
```

Usage du langage assembleur

Il y a des débats sur l'utilité du langage assembleur. Dans beaucoup de cas, des compilateurs-optimiseurs peuvent transformer du langage de haut niveau en un code qui tourne aussi efficacement qu'un code assembleur écrit à la main par un très bon programmeur, tout en restant beaucoup plus facile, rapide (et donc moins coûteux) à écrire, à lire et à maintenir.

L'efficacité était déjà une préoccupation dans les années 1950, on en trouve trace dans le manuel du langage Fortran (sorti en 1956) pour l'ordinateur IBM 704 :

Object programs produced by Fortran will be nearly as efficient as those written by good programmers.

Les compilateurs ayant entre-temps fait d'énormes progrès, il est donc évident que l'immense majorité des programmes sont maintenant écrits en langages de haut niveau pour des raisons économiques, le surcoût de programmation l'emportant très largement sur le gain résultant de l'amélioration espérée des performances.

Cependant, il reste quelques cas très spécifiques où l'utilisation de l'assembleur se justifie encore :

1. quelques calculs complexes écrits directement en assembleur, en particulier sur des machines massivement parallèles, seront plus rapides, les compilateurs n'étant pas assez évolués pour tirer parti des spécificités de ces architectures ;
2. certaines routines (*drivers*) sont parfois plus simples à écrire en langage de bas niveau ;
3. des tâches très dépendantes du système, exécutées dans l'espace mémoire du système d'exploitation sont parfois difficiles, voire impossibles à écrire dans un langage de haut niveau. Par exemple, les instructions assembleur qui permettent à Windows de gérer le changement de tâche (LGDT et LLDT) sur microprocesseur i386 et suivants ne peuvent pas être émulées ou générées par un langage évolué. Il faut nécessairement les coder dans un court sous-programme assembleur qui sera appelé à partir d'un programme écrit en langage évolué.

Certains compilateurs transforment, *lorsque leur option d'optimisation la plus haute n'est pas activée*, des programmes écrits en langage de haut niveau en code assembleur, chaque instruction de haut niveau se traduisant en une série d'instructions assembleur rigoureusement équivalentes et utilisant les mêmes symboles ; cela permet de voir le code dans une optique de débogage et de *profilage*, ce qui permet de gagner parfois beaucoup plus de temps en remaniant un algorithme. En aucun cas ces techniques ne peuvent être conservées pour l'optimisation finale.

La programmation des systèmes embarqués, souvent à base de microcontrôleurs, est une « niche » traditionnelle pour la programmation en assembleur. En effet ces systèmes sont souvent très limités en ressources (par exemple un microcontrôleur PIC 16F84 est limité à 1024 instructions de 14 bits, et sa mémoire vive contient 136 octets) et requièrent donc une programmation de bas-niveau très optimisée pour en exploiter les possibilités. Toutefois, l'évolution du matériel fait que les composants de ces systèmes deviennent de plus en plus puissants à un coût et à une consommation électrique constants, l'investissement dans une programmation « tout assembleur » beaucoup plus coûteuse en heures de travail devient alors un non-sens en termes d'efforts. Typiquement, la programmation en assembleur est beaucoup plus longue, plus délicate (car le programmeur doit prendre en compte tous les micro-détails du développement dont il s'abstient en langage évolué) et donc considérablement plus coûteuse que la programmation en langage de haut niveau. Il ne faut donc la réserver qu'aux situations pour lesquelles on ne peut pas faire autrement.

Macro-assembleur

Beaucoup d'assembleurs gèrent un langage de macros. Il s'agit de regrouper plusieurs instructions afin d'avoir un enchaînement plus logique et moins fastidieux.
Par exemple (en assembleur Microsoft MASM) :

```
putchar Macro   car          ; Prototype de la macro
        ifdef   car          ; si car est défini
        mov     dl,car       ;   le mettre dans dl
        endif
        mov     ah,2         ; ah=2 : fonction "putchar" en DOS
        int     21h          ; appel au DOS
        endm                 ; fin macro
```

est une macro qui affiche un caractère sous MS-DOS. On l'utilisera par exemple ainsi :

```
      putchar "X"
```

Et cela générera :

```
      mov    dl,"X"
      mov    ah,2
      int    21h
```

Pseudo-instructions

Une pseudo-instruction est un type particulier de macro-instruction. Elle est prédéfinie par l'éditeur du logiciel assembleur et sa fonction est d'émuler une instruction manquante du processeur ou de faciliter l'usage d'une instruction existante. Comme la pseudo-instruction a un nom très ressemblant à celui d'une vraie instruction du processeur, il est possible à première vue de la confondre avec une de ces dernières. Par exemple, un processeur RISC peut ne pas posséder d'instruction JMP, instruction permettant de sauter à un point particulier du programme et de continuer son exécution en séquence. L'éditeur du logiciel aura dans ce cas créé à l'intention du programmeur une pseudo-instruction « JMP <paramètre> », qui sera remplacée à l'assemblage par une instruction « mov *pc*, <paramètre> », *pc* étant le pointeur de l'instruction sur le point d'être exécutée. Autre exemple, une pseudo-instruction « PUSH <paramètre> » sera remplacée par un stockage de <paramètre> à l'adresse pointée par *sp* avec pré-décrémentation de celui-ci, *sp* étant le pointeur de pile du processeur.

Sur des microprocesseurs ou microcontrôleurs RISC tels que ceux de la famille ARM, il n'existe pas d'instruction assembleur permettant de charger n'importe quelle constante immédiate dans un registre, quelle que soit sa valeur. La plupart des assembleurs disposent d'une pseudo-instruction permettant un tel chargement de la façon la plus efficace possible en termes de temps d'exécution, épargnant cette tâche au programmeur

Programmation structurée en assembleur

Support pour programmation structurée : quelques éléments de programmation structurée ont été intégrés pour encoder le flux d'exécution par le docteur H. D. Mills (mars 1970), et mis en œuvre par Marvin Kessler qui a étendu l'assembleur S/360 macro avec if / else / endif et même des blocs de contrôle de flux. Cela a été un moyen de réduire ou d'éliminer l'utilisation des opérations de sauts dans le code assembleur

Notes et références

1. Laurent Bloch, *Initiation à la programmation avec Scheme*, Éditions TECHNIP, 2011 (lire en ligne (https://books.google.fr/books?id=8ogrKMfpoywC&pg=PA20&dq=%22Assembleur%22+langage+1949&hl=fr&sa=X&ved=0ahUKEwia5JfCnIbLAhWBRBoKHaVFBnsQ6AEIHDAA#v=onepage&q=%22Assembleur%22%20langage%201949&f=false))

Voir aussi

- Programme assembleur

Sur les autres projets Wikimedia :

- *Assembleur*, sur Wikimedia Commons
- *assembleur*, sur le Wiktionnaire
- *Assembleur*, sur Wikiversity
- *Assembleur*, sur Wikibooks